Chapter 5 An Introduction to Vector Searching and Sorting

Searching and sorting are two of the most frequently performed computing tasks. In this chapter we will examine several elementary searching and sorting algorithms in the interests of applying the vector ADT and of introducing asymptotic algorithm analysis. We will briefly revisit sorting in Chapter 6, and we will take another more intensive look at sorting in Chapter 9, after having developed the tools to understand and implement the more sophisticated and better-performing algorithms presented there.

5.1 Our Current Model

Our current class diagram appears below in Figure 5.1. In this chapter we add classes `Search` and `Sort`, both of which depend on class `Vector`.

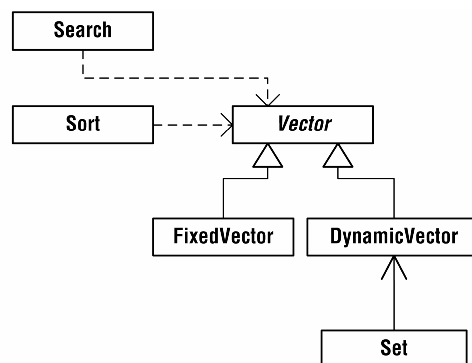

Fig. 5.1. Our current model; a `Set` *has-a* `DynamicVector`, and classes `Search` and `Sort` depend on class `Vector`

5.2 Asymptotic Algorithm Analysis

The goal of *asymptotic algorithm analysis* is to arrive at an expression for the growth rate of an algorithm's running time or space requirement in terms of the algorithm's input size. Such expressions allow us to compare the performance of two algorithms independent of any particular computing hardware. In other words, asymptotic algorithm analysis helps us to choose among two or more algorithms designed to solve the same problem without having to implement the algorithms and compare their performance by executing them on a particular computer. Asymptotic algorithm analysis facilitates the *formal* comparison of algorithms, making *empirical* comparisons unnecessary.

5.2.1 Growth Rates

The key concept in asymptotic algorithm analysis is that of a *growth rate*.

growth rate: a mathematical expression that tells us how rapidly an algorithm's running time or space requirement increases as the problem size increases

Graphs of some of the growth rates most commonly encountered in the computer science literature appear below in Figure 5.2. The input size or problem size is denoted n. The base-2 logarithm function is denoted lg.

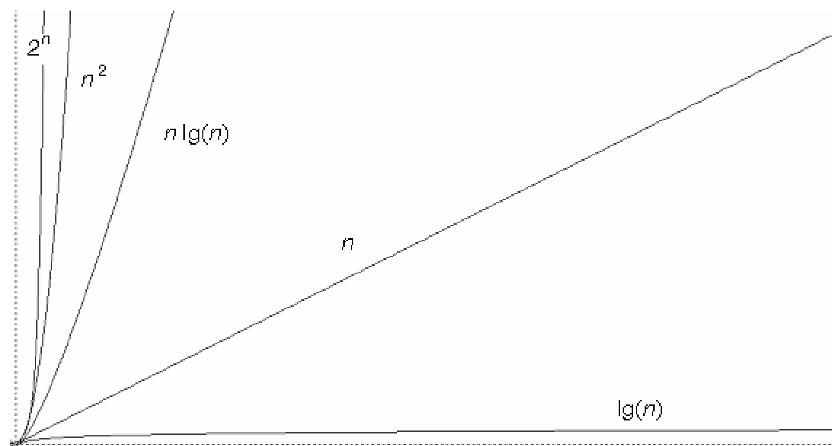

Fig. 5.2. Algorithmic Growth Rates Commonly Encountered in the Computer Science Literature

The lg(n) growth rate is also called *logarithmic*, the n growth rate is also called *linear*, the n^2 growth rate is also called *quadratic*, and the 2^n growth rate is also called *exponential*. We can see that an algorithm whose running time or space requirement grows logarithmically is preferred to an algorithm whose running time or space requirement grows exponentially. Increasing an exponential algorithm's input size by one causes the running time or space requirement to double, while doubling a logarithmic algorithm's input size causes its running time or space requirement to increase by just one!

5.2.2 Putting Growth Rates to Work: The Bubble Sort Algorithm and Its Time Analysis

Let us step through the bubble sort algorithm on the array of integers shown below.

7	-2	4	9	18	23	0	9	3
0	1	2	3	4	5	6	7	8

On its first pass through the data the algorithm begins by comparing the last item, 3, with its predecessor, 9. Because those two items are not in ascending order they are swapped, yielding the array shown below.

7	-2	4	9	18	23	0	3	9
0	1	2	3	4	5	6	7	8

Next the item at index 7 is compared with the item at index 6. Those two items are in ascending order, and so no swap is necessary. The item at index 6 is then compared with the item at index 5. The 0 and the 23 are out of order, and so they are swapped, yielding the array shown below.

7	-2	4	9	18	0	23	3	9
0	1	2	3	4	5	6	7	8

The algorithm next compares the items at indices 5 and 4, after which the two values are swapped. The resulting array appears below.

7	-2	4	9	0	18	23	3	9
0	1	2	3	4	5	6	7	8

The comparison of the items at indices 4 and 3 is followed by another swap, and the subsequent comparison of the items at indices 3 and 2 results in yet another swap, yielding the array shown below.

7	-2	0	4	9	18	23	3	9
0	1	2	3	4	5	6	7	8

Because the items at indices 2 and 1 are in ascending order no swap occurs. But the -2 and the 7 are out of order and must be swapped. We are left with the following array.

-2	7	0	4	9	18	23	3	9
0	1	2	3	4	5	6	7	8

This concludes the algorithm's first pass. The next pass begins in the same fashion but need not traverse the entire array, for the -2 has reached its final resting place. After the second pass the 0 has reached its final resting place and the array is in the state shown below.

-2	0	7	3	4	9	18	23	9
0	1	2	3	4	5	6	7	8

Because the -2 and the 0 have reached their final destinations the algorithm's third pass must examine only locations 2 through 8. After the third pass the 3 has been placed at index 2 and the array is in the state shown below.

-2	0	3	7	4	9	9	18	23
0	1	2	3	4	5	6	7	8

Subsequent passes proceed in the same fashion. On its final pass the algorithm deals only with the items at indices 7 and 8. The result is of course a sorted array.

In analyzing the algorithm, we consider three cases: best, average, and worst; and we count the number of comparisons and the number of swaps because comparing and swapping are the algorithm's two fundamental operations.

Counting Comparisons

Observe that for an input size of n the bubble sort algorithm performs n - 1 passes. The algorithm does n - 1 comparisons on the first pass, n - 2 comparisons on the second pass, n - 3 comparisons on the third pass, ... , two comparisons on the next-to-last pass, and one comparison on the final pass. Thus the number of comparisons required when the input size is n is equal to

$$(n-1)+(n-2)+(n-3)+...+3+2+1=\frac{n(n-1)}{2}=\tfrac{1}{2}n^2-\tfrac{1}{2}n.$$

This is true in the best, average, and worst cases.

Counting Swaps

In the best case, i.e., when the data are already sorted, the bubble sort algorithm does no swapping. In the worst case, i.e., when the data are sorted in descending order, the algorithm must swap each time it compares. Thus the number of swaps in the worst case is $\tfrac{1}{2}n^2-\tfrac{1}{2}n$.

We may safely assume that the algorithm must swap after every other comparison, or about half the time, on average. Thus the number of swaps in the average case is $\tfrac{1}{4}n^2-\tfrac{1}{4}n$.

Summing our results, we arrive at the following.

Table 5.1. Best-, average-, and worst-case running times for the bubble sort algorithm

Case	Comparisons	Swaps	Total Operations
Best	$\tfrac{1}{2}n^2-\tfrac{1}{2}n$	0	$\tfrac{1}{2}n^2-\tfrac{1}{2}n$
Average	$\tfrac{1}{2}n^2-\tfrac{1}{2}n$	$\tfrac{1}{4}n^2-\tfrac{1}{4}n$	$\tfrac{3}{4}n^2-\tfrac{3}{4}n$
Worst	$\tfrac{1}{2}n^2-\tfrac{1}{2}n$	$\tfrac{1}{2}n^2-\tfrac{1}{2}n$	n^2-n

The Final Product: Big-Omicron Expressions

Consider the expression for the total number of operations performed by the bubble sort algorithm in the best case. The quadratic term clearly dominates the linear term. The graph of $\frac{1}{2}n^2 - \frac{1}{2}n$ has the same basic shape as the graph of n^2, and the shape is what concerns us, not the particular value we get upon evaluating the expression at a particular value of n. Consequently, we discard the linear term, which leaves us with $\frac{1}{2}n^2$.

The coefficient, $\frac{1}{2}$, also does not affect the basic shape of the graph, and so the coefficient may also be discarded, leaving us with n^2. We likewise discard the linear terms and the coefficients from the average-case and worst-case expressions, arriving at n^2 for those cases, as well. We express the results of our analysis using *big-Omicron notation*.

big-Omicron (big-Oh) notation: if the growth rate of algorithm A's running time or space requirement is no worse than $f(n)$, then we say that A is in $O(f(n))$, i.e., a big-Omicron expression gives an *upper bound* on an algorithm's growth rate; we use the term *in* because $f(n)$ may not be a tight upper bound for A, i.e., the actual upper bound on A's growth rate may be smaller than $f(n)$

The running time of the bubble sort algorithm is in $O(n^2)$ in the best, average, and worst cases.

5.2.3 Another Example: The Selection Sort Algorithm and Its Time Analysis

We will apply the selection sort algorithm to the same array.

7	-2	4	9	18	23	0	9	3
0	1	2	3	4	5	6	7	8

On its first pass the algorithm finds the smallest value in the array, -2, and then swaps that value with the value at index 0. This gives the array shown below.

-2	7	4	9	18	23	0	9	3
0	1	2	3	4	5	6	7	8

Because the smallest value is now in its final resting place, the algorithm need not examine location 0 on its second pass. The algorithm scans locations 1 through 8 and finds the smallest value among those locations, 0. The 0 is then swapped with the item at index 1, yielding the following array.

-2	0	4	9	18	23	7	9	3
0	1	2	3	4	5	6	7	8

On its third pass the selection sort need not examine locations 0 and 1. Locations 2 through 8 are scanned and 3 is found to be the smallest value among those locations. The 3 is swapped with the value at index 2. The resulting array appears below.

-2	0	3	9	18	23	7	9	4
0	1	2	3	4	5	6	7	8

On its fourth pass the algorithm swaps the 4 with the value at index 3, arriving at the following array.

-2	0	3	4	18	23	7	9	9
0	1	2	3	4	5	6	7	8

This process continues until the array is sorted.

Time Analysis of the Selection Sort

We again count comparisons and swaps for the best, average, and worst cases. The number of comparisons performed in each case is the same as for the bubble sort algorithm, $\frac{1}{2}n^2 - \frac{1}{2}n$. In the best case, i.e., when the data are already in ascending order, the selection sort algorithm never swaps. In the worst case, i.e., when the data are in descending order, the algorithm must swap once on each pass, or n - 1 times. In the average case the algorithm must swap half as often, or $\frac{1}{2}n - \frac{1}{2}$ times. Summing our results for each case, we arrive at the following.

Table 5.2. Best-, average-, and worst-case running times for the selection sort algorithm

Case	Comparisons	Swaps	Total Operations
Best	$\frac{1}{2}n^2 - \frac{1}{2}n$	0	$\frac{1}{2}n^2 - \frac{1}{2}n$
Average	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n - \frac{1}{2}$	$\frac{1}{2}n^2 - \frac{1}{2}$
Worst	$\frac{1}{2}n^2 - \frac{1}{2}n$	n - 1	$\frac{1}{2}n^2 + \frac{1}{2}n - 1$

We discard the constant and linear terms and coefficients and are left with n^2 for each of the three cases. Thus the selection sort algorithm is in $O(n^2)$ in the best, average, and worst cases.

5.3 Vector Searching: The Linear and Binary Search Algorithms

We will now examine two searching algorithms, the linear search and the binary search.

5.3.1 The Linear Search Algorithm

The linear search algorithm amounts to scanning the elements of a linear collection, in our case a vector, until a target element is found or is determined to be absent from the collection. If the target element is the first element, then only one comparison is required. In the average case about one-half of the elements must be examined before the target element is found, i.e., $\frac{1}{2}n$ comparisons must be performed in the average case.

The linear search algorithm's worst-case performance occurs when the target element is not a member of the collection. The number of comparisons required in this case depends on whether the collection is sorted. If the collection is not sorted, then all n elements must be examined in order to determine that the target is absent. If the collection is sorted, then about half of the elements must be examined, on average, to determine that the target is absent. These results are summarized in the table below.

Table 5.3. Best-, average-, and worst-case running times for the linear search algorithm

Algorithm	Case	Comparisons
Linear search, sorted collection	Best	1
	Average	$\frac{1}{2}n$
	Worst	$\frac{1}{2}n$
Linear search, unsorted collection	Best	1
	Average	$\frac{1}{2}n$
	Worst	n

We see that both algorithms are in O(1) in the best case and in O(n) in the average and worst cases. An algorithm that is in O(1) is called a *constant-time algorithm*. The running time or space requirement of a constant-time algorithm does not depend on input size.

5.3.2 The Binary Search Algorithm

The binary search algorithm depends on a sorted collection. We will apply the algorithm to the array shown below. Let us search for 65.

-2	0	3	4	7	9	9	18	23	32	65	70
0	1	2	3	4	5	6	7	8	9	10	11

Algorithm binary search first finds the middle element by adding the first and last indices and integer dividing the result by 2. Given our example array, this computation yields (11 + 0) / 2 = 5. Thus the middle element is at index 5. The target element, 65, is compared to the middle element, 9. Because 65 > 9 and the collection is sorted, the target value must occupy one of locations 6 through 11, if it is present at all. Consequently, the subcollection at indices 0 through 5 may be excluded from further consideration.

9	18	23	32	65	70
6	7	8	9	10	11

Now the algorithm, having discarded the first half of the collection, finds the middle element among locations 6 through 11: (11 + 6) / 2 = 8. The element at index 8 is compared with the target element.

Because 65 > 23 and the collection is sorted, if the target element is present it must occupy one of locations 9 through 11, and so the subcollection at indices 6 through 8 may be discarded.

32	65	70
9	10	11

The middle element of this subcollection is at index (11 + 9) / 2 = 10. The element at index 10 is the target element, and so the search is complete. If the target element had been absent, then the algorithm would have terminated after having "homed in" on a one-element subcollection.

The binary search algorithm is an example of a *divide-and-conquer algorithm*.

divide-and-conquer algorithm: an algorithm that solves a problem by repeatedly reducing the problem size

5.4 Implementing the Aforementioned Algorithms: Classes `Search` and `Sort`

Java implementations of the bubble sort, selection sort, linear search, and binary search are shown below. The implementations assume that any instance of `Vector` to which an algorithm is being applied contains elements that can be ordered. Instances of any class that implements Java's `Comparable` interface satisfy this condition.

5.4.1 Java Interfaces and Interface `Comparable`

The Java programming language provides a facility for creating ADT specifications, the *interface*. The term *interface* is both a UML term and a Java term. In either context the term refers to an ADT. A UML interface is represented in the manner of `Cloneable` from Figure 1.6. A Java interface is written in the Java programming language. The code for Java interface `Comparable` appears below.

```
public interface Comparable
{
    int compareTo(Object o);
}
```

Observe that interface `Comparable` contains no implementation. This is in keeping with the fact that interfaces are for specifying ADTs. Recall that an ADT does not specify implementation details; an ADT specifies only *what* a data type can do, not *how* it is to be done. A data structure provides the *how*.

What, then, is the purpose of a Java interface? A Java interface may be thought of as a sort of contract. Any instantiable Java class that realizes an interface must implement each of the interface's methods. A Java class that fails to do so will not compile.

Any class that implements interface `Comparable` offers a public method called `compareTo`. This method compares the calling object, `this`, with object `o`. If `this` is less than `o`, then `compareTo` returns a negative value. If `this` is equal to `o`, then `compareTo` returns 0. If `this` is greater than `o`, then `compareTo` returns a positive value. Thus instances of any class that implements `Comparable` can be ordered.

5.4.2 Classes `Search` and `Sort`

The linear and binary search algorithms are implemented as class methods of class `Search`, and the bubble and selection sort algorithms are implemented as class methods of class `Sort`. Each method takes an argument of type `Vector`. Recall that `Vector` is an abstract class with concrete subclasses `FixedVector` and `DynamicVector`. Because `FixedVector` is *ako* `Vector` and `DynamicVector` is *ako* `Vector`, the actual type of the container passed to any of the methods may be either `FixedVector` or `DynamicVector`. In other words, the inheritance implemented in Chapter 4 makes `FixedVectors` and `DynamicVectors` interchangeable as far as classes `Search` and `Sort` are concerned.

```
public class Search
{
    private Search() {}  // Prevent instantiation.

    public static int binarySearch(Vector vec, Comparable target)
    {
        int first = 0,
            last = vec.size() - 1,
            middle;
        while (last - first >= 0)
        {
            middle = (first + last) / 2;
            if (target.compareTo(vec.elementAt(middle)) < 0)
                last = middle - 1;
            else if (target.compareTo(vec.elementAt(middle)) > 0)
                first = middle + 1;
            else
                return middle;
        }
        return -1;
    }

    public static int linearSearchSorted(Vector vec,
                                         Comparable target)
    {
        int j,
            n = vec.size();
        for (j = 0;
             j < n && target.compareTo(vec.elementAt(j)) > 0;
             j++);
        if (j < n && target.compareTo(vec.elementAt(j)) == 0)
            return j;
        return -1;
    }
}

public class Sort
{
    private Sort() {}  // Prevent instantiation.

    public static void swap(Vector vec, int first, int second)
    {
        Object temp = vec.elementAt(first);
        vec.replace(first, vec.elementAt(second));
        vec.replace(second, temp);
    }

    public static void bubbleSort(Vector vec)
    {
        int i,
            j,
            n = vec.size();
        Comparable first,
                   second;
        for (i = 1; i < n; i++)
            for (j = n - 1; j >= i; j--)
            {
                first = (Comparable)vec.elementAt(j - 1);
                second = (Comparable)vec.elementAt(j);
                if (first.compareTo(second) > 0)
                    swap(vec, j - 1, j);
            }
    }
```

```
    public static void selectionSort(Vector vec)
    {
        int i,
            j,
            n = vec.size(),
            smallPos;
        Comparable smallest,
                   current;
        for (i = 0; i < n - 1; i++)
        {
            smallPos = i;
            smallest = (Comparable)vec.elementAt(smallPos);
            for (j = i + 1; j < n; j++)
            {
                current = (Comparable)vec.elementAt(j);
                if (current.compareTo(smallest) < 0)
                {
                    smallPos = j;
                    smallest =
                        (Comparable)vec.elementAt(smallPos);
                }
            }
            if (smallPos != i)
                swap(vec, i, smallPos);
        }
    }
}
```

Exercises

1. Add attributes and operations to the class diagram of Figure 5.1.
2. According to the vector ADT, a vector is inherently ordered, yet we have just examined two algorithms designed to order a vector. Explain.
3. The bubble sort and selection sort algorithms are both in $O(n^2)$ in the average and worst cases, yet selection sort performs better in both cases. Explain.
4. Perform a time analysis of the binary search algorithm. Be sure to consider the special case where n is a power of 2.
5. The linear search of an unsorted `Vector` does not appear in class `Search`. Why not?
6. How is the behavior of method `compareTo` related to the signum mathematical function?
7. Comment the methods of `Search` and `Sort` using the precondition/postcondition format illustrated in Chapters 3 and 4.
8. What will happen if one of the searching or sorting methods of `Search` or `Sort` encounters an instance of some type that is not `Comparable`? Should interface `Comparable` appear in our model? If `Comparable` should be included in our model, put it there. Otherwise explain why it should not be included.